

CS 145 Midterm Review

The Best Of Collection (Master Tracks), Vol. 1

High-Level: Lecture 2

- Basic terminology:
 - relation / table (+ “instance of”), row / tuple, column / attribute, multiset
- Table schemas in SQL
- Single-table queries:
 - SFW (selection + projection)
 - Basic SQL operators: LIKE, DISTINCT, ORDER BY
- Multi-table queries:
 - Foreign keys
 - JOINS:
 - Basic SQL syntax & semantics of

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	\$19.99	GizmoWorks
Powergizmo	\$29.99	GizmoWorks
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

A relation or table is a multiset of tuples having the attributes specified by the schema

A multiset is an unordered list (or: a set with multiple duplicate instances allowed)

A tuple or row is a single entry in the table having the attributes specified by the schema

An attribute (or column) is a typed data entry present in each tuple in the relation

Table Schemas

- The **schema** of a table is the table name, its attributes, and their types:

```
Product(Pname: string, Price: float, Category:  
string, Manufacturer: string)
```

- A **key** is an attribute whose values are unique; we underline a key

```
Product(Pname: string, Price: float, Category:  
string, Manufacturer: string)
```


SQL Query

- Basic form (there are many many more bells and whistles)

```
SELECT <attributes>  
FROM   <one or more relations>  
WHERE  <conditions>
```

Call this a SFW query.

LIKE: Simple String Pattern Matching

```
SELECT *  
FROM Products  
WHERE PName LIKE '%gizmo%'
```

DISTINCT: Eliminating Duplicates

```
SELECT DISTINCT Category  
FROM Product
```

ORDER BY: Sorting the Results

```
SELECT PName, Price  
FROM Product  
WHERE Category='gizmo'  
ORDER BY Price, PName
```

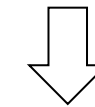
Joins

Product

PName	Price	Category	Manuf
Gizmo	\$19	Gadgets	GWorks
Powergizmo	\$29	Gadgets	GWorks
SingleTouch	\$149	Photography	Canon
MultiTouch	\$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

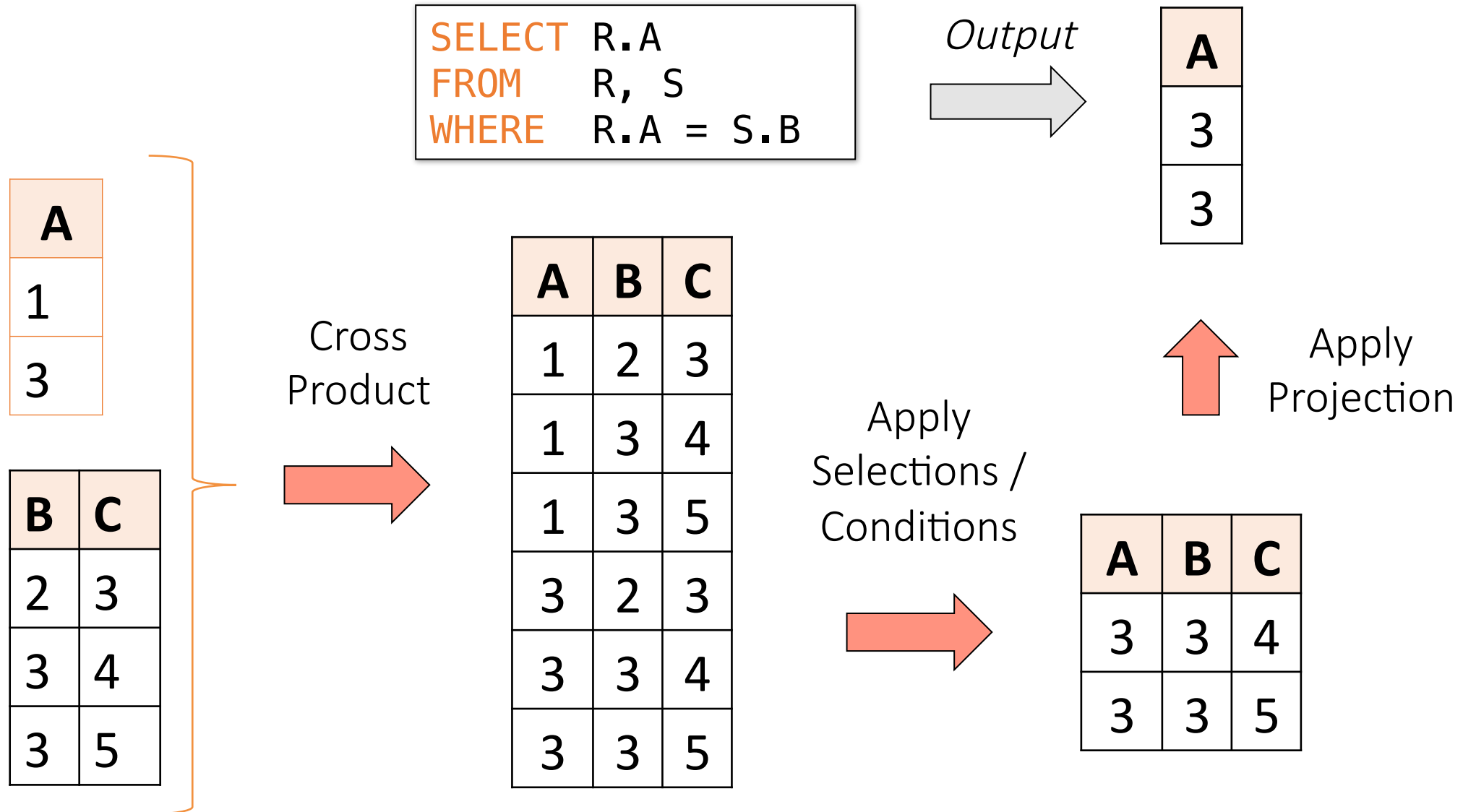


```

SELECT PName, Price
FROM Product, Company
WHERE Manufacturer = CName
      AND Country='Japan'
      AND Price <= 200
  
```

PName	Price
SingleTouch	\$149.99

An example of SQL semantics

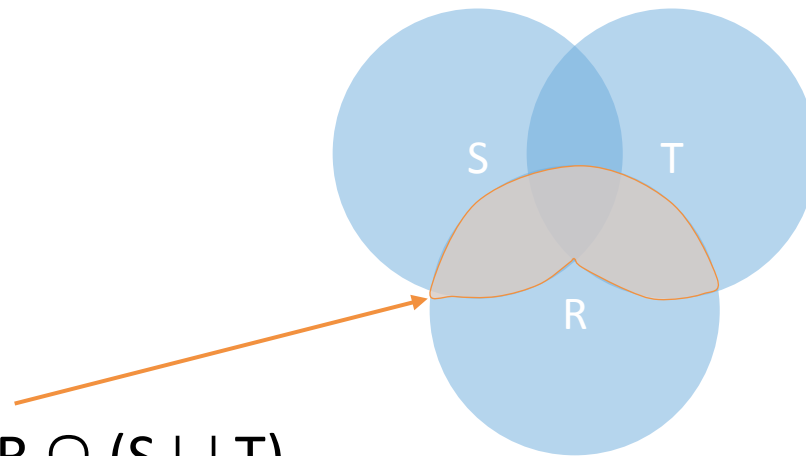


High-Level: Lecture 3

- Set operators
 - INTERSECT, UNION, EXCEPT, [ALL]
 - Subtleties of multiset operations
- Nested queries
 - IN, ANY, ALL, EXISTS
 - Correlated queries
- Aggregation
 - AVG, SUM, COUNT, MIN, MAX, ...
- GROUP BY
- NULLs & Outer Joins

An Unintuitive Query

```
SELECT DISTINCT R.A  
FROM   R, S, T  
WHERE  R.A=S.A OR R.A=T.A
```



Computes $R \cap (S \cup T)$

But what if $S = \phi$?

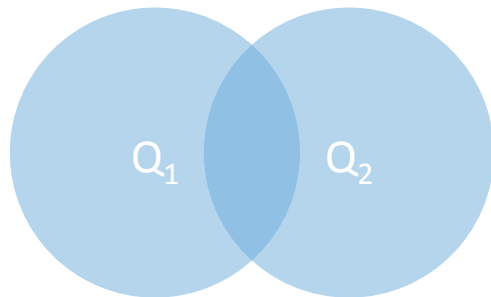
Go back to the semantics!

INTERSECT

```

SELECT R.A
FROM   R, S
WHERE  R.A=S.A
INTERSECT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A

```

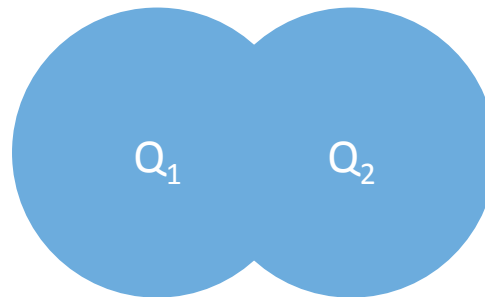


UNION

```

SELECT R.A
FROM   R, S
WHERE  R.A=S.A
UNION
SELECT R.A
FROM   R, T
WHERE  R.A=T.A

```

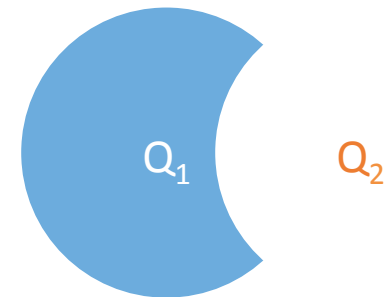


EXCEPT

```

SELECT R.A
FROM   R, S
WHERE  R.A=S.A
EXCEPT
SELECT R.A
FROM   R, T
WHERE  R.A=T.A

```



Nested queries: Sub-queries Returning Relations

```
Company(name, city)
Product(name, maker)
Purchase(id, product, buyer)
```

```
SELECT c.city
FROM   Company c
WHERE  c.name IN (
        SELECT pr.maker
        FROM   Purchase p, Product pr
        WHERE  p.product = pr.name
              AND p.buyer = 'Joe Blow')
```

“Cities where one can find companies that manufacture products bought by Joe Blow”

Nested Queries: Operator Semantics

Product(name, price, category, maker)

ALL

```
SELECT name
FROM Product
WHERE price > ALL(
    SELECT price
    FROM Product
    WHERE maker = 'G')
```

Find products that are more expensive than *all products* produced by “G”

ANY

```
SELECT name
FROM Product
WHERE price > ANY(
    SELECT price
    FROM Product
    WHERE maker = 'G')
```

Find products that are more expensive than *any one product* produced by “G”

EXISTS

```
SELECT name
FROM Product p1
WHERE EXISTS (
    SELECT *
    FROM Product p2
    WHERE p2.maker = 'G'
    AND p1.price = p2.price)
```

Find products where *there exists some* product with the same price produced by “G”

Nested Queries: Operator Semantics

Product(name, price, category, maker)

ALL

```
SELECT name
FROM   Product
WHERE  price > ALL(X)
```

Price must be > *all* entries
in multiset X

ANY

```
SELECT name
FROM   Product
WHERE  price > ANY(X)
```

Price must be > *at least
one* entry in multiset X

EXISTS

```
SELECT name
FROM   Product p1
WHERE  EXISTS (X)
```

X must be non-empty

**Note that p1 can be
referenced in X
(correlated query!)*

Correlated Queries

```
Movie(title, year, director, length)
```

```
SELECT DISTINCT title
FROM   Movie AS m
WHERE  year <> ANY(
        SELECT year
        FROM   Movie
        WHERE  title = m.title)
```

Find movies whose title appears more than once.

Note the scoping of the variables!

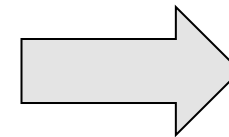
Note also: this can still be expressed as single SFW query...

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

```
SELECT SUM(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```



50 (= 1*20 + 1.50*20)

Grouping & Aggregations: GROUP BY

```
SELECT    product, SUM(price*quantity)
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY  product
HAVING    SUM(quantity) > 10
```

Find total sales after
10/1/2005, only for
products that have
more than
10 total units sold

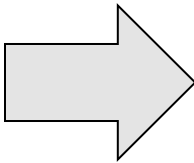
HAVING clauses contains conditions on aggregates

Whereas WHERE clauses condition on individual tuples...

GROUP BY: (1) Compute FROM-WHERE

```
SELECT product, SUM(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING SUM(quantity) > 10
```

FROM
WHERE



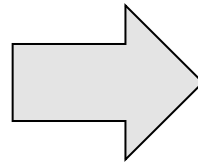
Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10
Craisins	11/1	2	5
Craisins	11/3	2.5	3

GROUP BY: (2) Aggregate by the GROUP BY

```
SELECT    product, SUM(price*quantity) AS TotalSales
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY product
HAVING    SUM(quantity) > 10
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10
Craisins	11/1	2	5
Craisins	11/3	2.5	3

GROUP BY



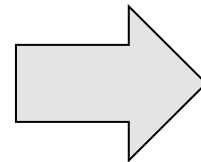
Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10
Craisins	11/1	2	5
	11/3	2.5	3

GROUP BY: (3) Filter by the **HAVING** clause

```
SELECT    product, SUM(price*quantity) AS TotalSales
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY  product
HAVING SUM(quantity) > 30
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10
Craisins	11/1	2	5
	11/3	2.5	3

HAVING



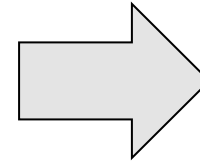
Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

GROUP BY: (3) **SELECT** clause

```
SELECT    product, SUM(price*quantity) AS TotalSales
FROM      Purchase
WHERE     date > '10/1/2005'
GROUP BY  product
HAVING    SUM(quantity) > 100
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT



Product	TotalSales
Bagel	50
Banana	15

General form of Grouping and Aggregation

SELECT	S
FROM	$R_1, \dots, R_n$
WHERE	C_1
GROUP BY	$a_1, \dots, a_k$
HAVING	C_2

Evaluation steps:

1. Evaluate **FROM-WHERE**: apply condition C_1 on the attributes in $R_1, \dots, R_n$
2. **GROUP BY** the attributes $a_1, \dots, a_k$
3. Apply **HAVING** condition C_2 to each group (may have aggregates)
4. Compute aggregates in **SELECT**, S, and return the result

Null Values

- *For numerical operations, NULL -> NULL:*
 - If $x = \text{NULL}$ then $4*(3-x)/7$ is still NULL
- *For boolean operations, in SQL there are three values:*

FALSE	=	0
UNKNOWN	=	0.5
TRUE	=	1

- If $x = \text{NULL}$ then $x = \text{"Joe"}$ is UNKNOWN

Null Values

- $C1 \text{ AND } C2 = \min(C1, C2)$
- $C1 \text{ OR } C2 = \max(C1, C2)$
- $\text{NOT } C1 = 1 - C1$

```
SELECT *  
FROM Person  
WHERE (age < 25)  
      AND (height > 6 AND weight > 190)
```

Won't return e.g.
(age=20
height=NULL
weight=200)!

Rule in SQL: include only tuples that yield TRUE / 1.0

Null Values

Unexpected behavior:

```
SELECT *  
FROM Person  
WHERE age < 25  
      OR age >= 25
```

Some Persons are not included !



```
SELECT *  
FROM Person  
WHERE age < 25  
      OR age >= 25  
      OR age IS NULL
```

Now it includes all Persons!

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

RECAP: Inner Joins

By default, joins in SQL are “**inner joins**”:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product
      JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

Both equivalent:
Both INNER JOINS!

INNER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM   Product
       INNER JOIN Purchase
       ON Product.name = Purchase.prodName
```

Note: another equivalent way to write an
INNER JOIN!



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

LEFT OUTER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM   Product
LEFT OUTER JOIN Purchase
ON     Product.name = Purchase.prodName
```



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

General clarification: Sets vs. Multisets

- In theory, and in any more formal material, **by definition** all relations are **sets of tuples**
- In SQL, relations (i.e. tables) are **multisets**, meaning you can have duplicate tuples
 - We need this because intermediate results in SQL don't eliminate duplicates
- If you get confused: just state your assumptions & we'll be forgiving!

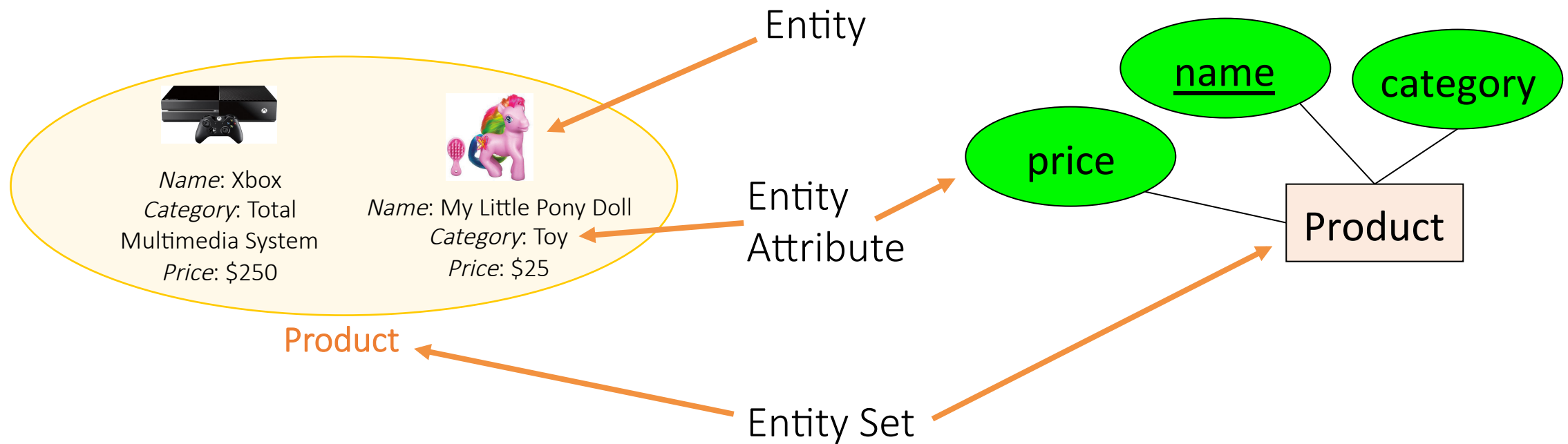
High-Level: Lecture 4

- ER diagrams!
 - Entities (vs. Entity Sets)
 - Relationships
 - Multiplicity
 - Constraints: Keys, single-value, referential, participation, etc...

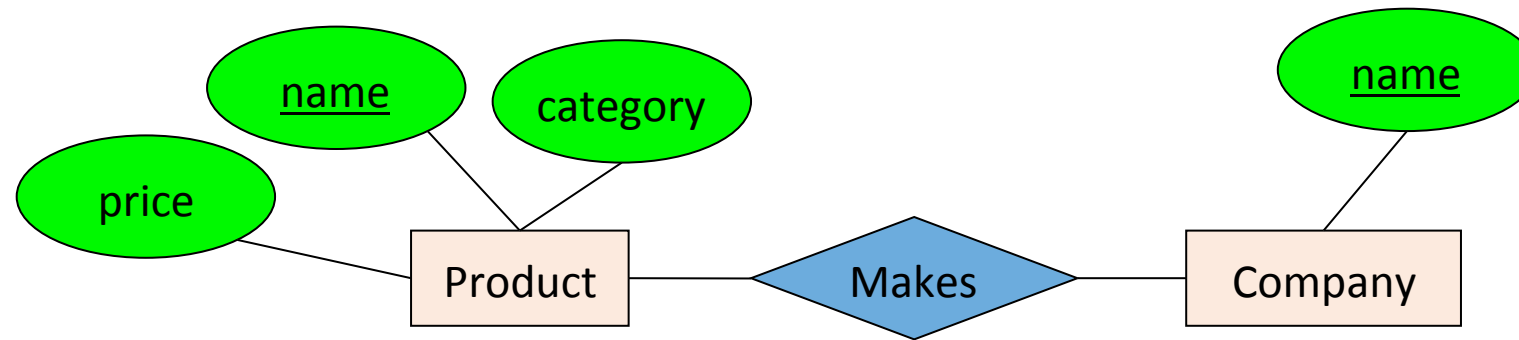
Entities vs. Entity Sets

Example:

Entities are not explicitly represented in E/R diagrams!



What is a Relationship?



A relationship between entity sets P and C is a *subset of all possible pairs of entities in P and C* , with tuples uniquely identified by *P and C 's keys*

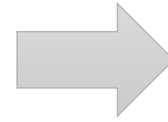
What is a Relationship?

Company

<u>name</u>
GizmoWorks
GadgetCorp

Product

<u>name</u>	category	price
Gizmo	Electronics	\$9.99
GizmoLite	Electronics	\$7.50
Gadget	Toys	\$5.50



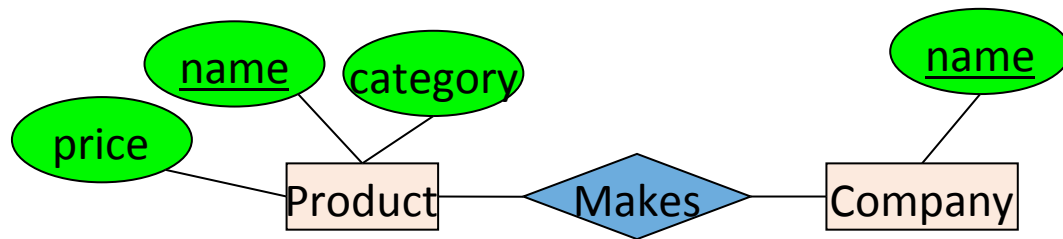
Company C × Product P

<u>C.name</u>	<u>P.name</u>	P.category	P.price
GizmoWorks	Gizmo	Electronics	\$9.99
GizmoWorks	GizmoLite	Electronics	\$7.50
GizmoWorks	Gadget	Toys	\$5.50
GadgetCorp	Gizmo	Electronics	\$9.99
GadgetCorp	GizmoLite	Electronics	\$7.50
GadgetCorp	Gadget	Toys	\$5.50



Makes

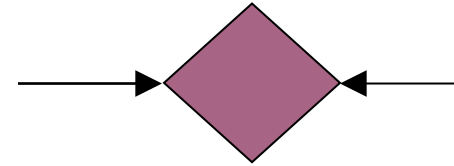
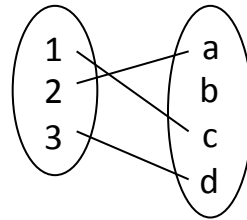
<u>C.name</u>	<u>P.name</u>
GizmoWorks	Gizmo
GizmoWorks	GizmoLite
GadgetCorp	Gadget



A relationship between entity sets P and C is a *subset of all possible pairs of entities in P and C*, with tuples uniquely identified by *P and C's keys*

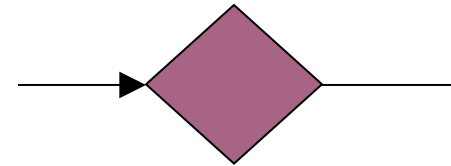
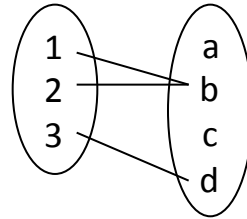
Multiplicity of E/R Relationships

One-to-one:

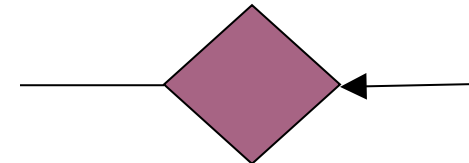
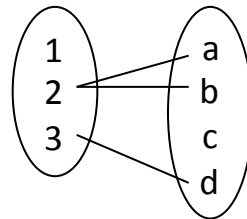


Indicated using
arrows

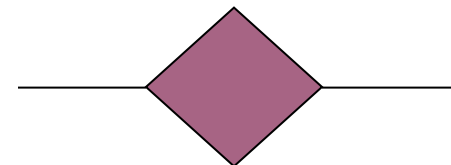
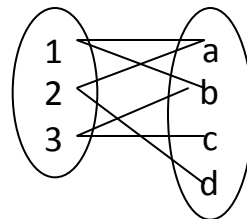
Many-to-one:



One-to-many:



Many-to-many:

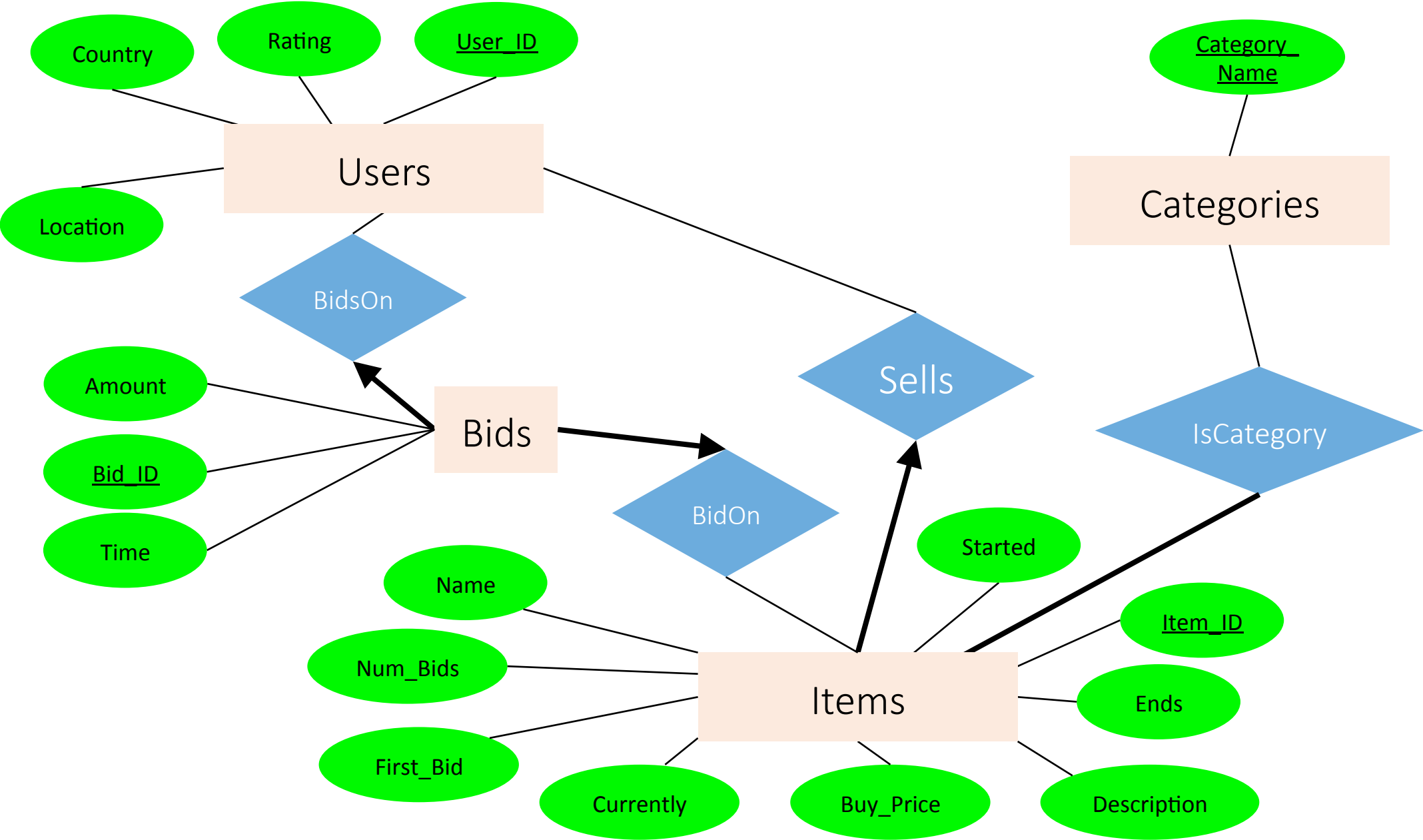


$X \rightarrow Y$ means
there exists a
function mapping
from X to Y (*recall*
the definition of a
function)

Constraints in E/R Diagrams

- Finding constraints is part of the E/R modeling process. Commonly used constraints are:
 - Keys: Implicit constraints on uniqueness of entities
 - *Ex: An SSN uniquely identifies a person*
 - Single-value constraints:
 - *Ex: a person can have only one father*
 - Referential integrity constraints: Referenced entities must exist
 - *Ex: if you work for a company, it must exist in the database*
 - Other constraints:
 - *Ex: peoples' ages are between 0 and 150*

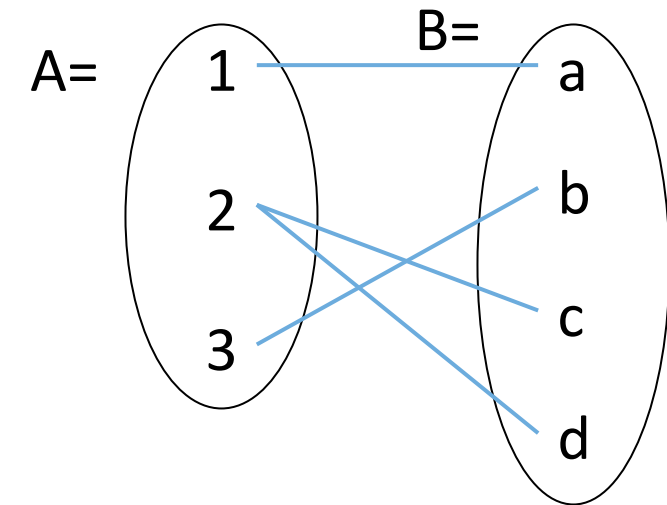
Recall
FOREIGN
KEYs!



RECALL: Mathematical def. of Relationship

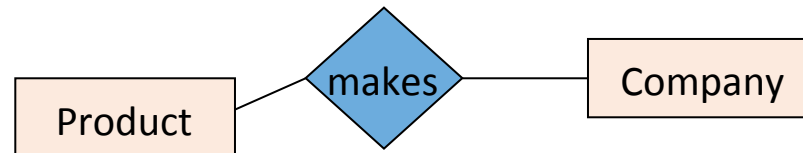
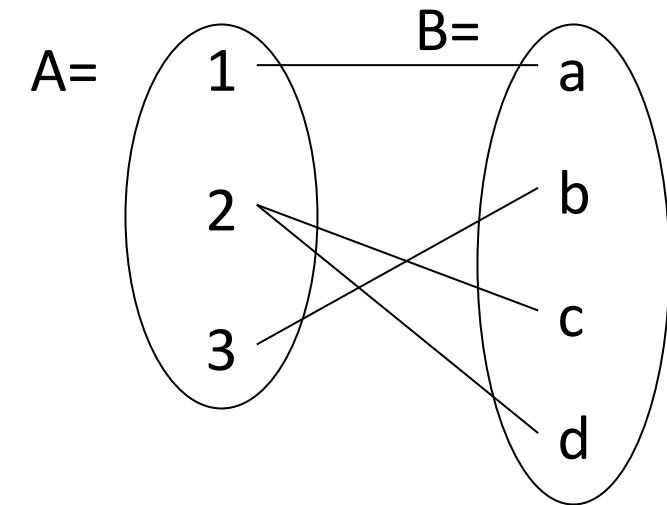
- ***A mathematical definition:***

- Let A, B be sets
 - $A=\{1,2,3\}$, $B=\{a,b,c,d\}$,
- $A \times B$ (the ***cross-product***) is the set of all pairs (a,b)
 - $A \times B = \{(1,a), (1,b), (1,c), (1,d), (2,a), (2,b), (2,c), (2,d), (3,a), (3,b), (3,c), (3,d)\}$
- We define a relationship to be a subset of $A \times B$
 - $R = \{(1,a), (2,c), (2,d), (3,b)\}$



RECALL: Mathematical def. of Relationship

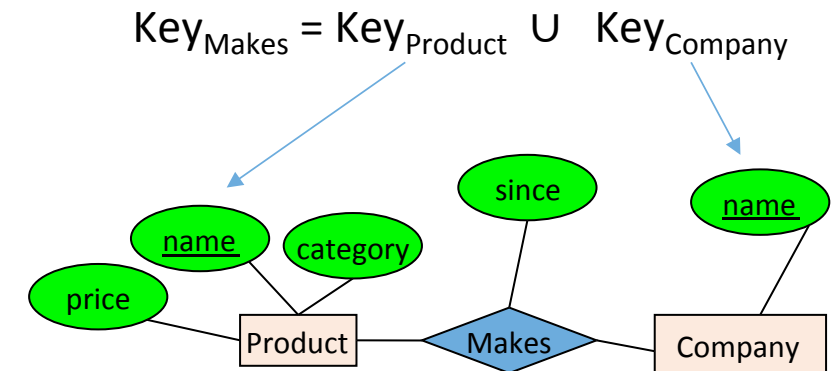
- ***A mathematical definition:***
 - Let A, B be sets
 - $A \times B$ (the ***cross-product***) is the set of all pairs
 - A relationship is a subset of $A \times B$
- **Makes** is relationship- it is a ***subset*** of **Product $\times$ Company**:



RECALL: Mathematical def. of Relationship

- There can only be **one relationship for every unique combination of entities**
- This also means that **the relationship is uniquely determined by the keys of its entities**
- *Example: the key for Makes (to right) is $\{Product.name, Company.name\}$*

This follows from our mathematical definition of a relationship- it's a SET!



Why does this make sense?

High-Level: Lecture 5

- Redundancy & data anomalies
- Functional dependencies
 - For database schema design
 - Given set of FDs, find others implied- using Armstrong's rules
- Closures
 - Basic algorithm
 - To find all FDs
- Keys & Superkeys

Constraints Prevent (some) Anomalies in the Data

A poorly designed database causes *anomalies*:

Similarly, we can't reserve a room without students = an insert anomaly

...	CS229	C12
-----	-------	-----



Student	Course	Room
Mary	CS145	B01
Joe	CS145	B01
Sam	CS145	B01
..	..	..

If every course is in only one room, contains redundant information!

If we update the room number for one tuple, we get inconsistent data = an update anomaly

If everyone drops the class, we lose what room the class is in! = a delete anomaly

Constraints Prevent (some) Anomalies in the Data

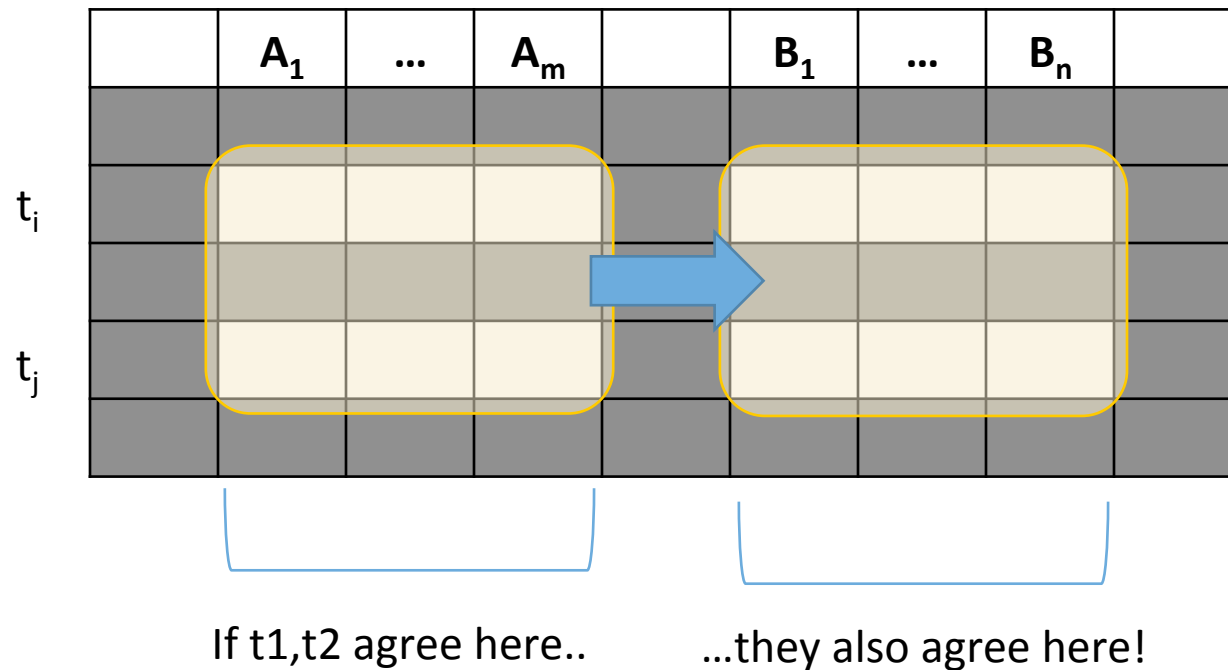
Student	Course
Mary	CS145
Joe	CS145
Sam	CS145
..	..

Course	Room
CS145	B01
CS229	C12

Is this form better?

- Redundancy?
- Update anomaly?
- Delete anomaly?
- Insert anomaly?

A Picture Of FDs



Defn (again):

Given attribute sets $A = \{A_1, \dots, A_m\}$ and $B = \{B_1, \dots, B_n\}$ in R ,

The *functional dependency* $A \rightarrow B$ on R holds if for *any* t_i, t_j in R :

if $t_i[A_1] = t_j[A_1]$ AND $t_i[A_2] = t_j[A_2]$ AND
... AND $t_i[A_m] = t_j[A_m]$

then $t_i[B_1] = t_j[B_1]$ AND $t_i[B_2] = t_j[B_2]$
AND ... AND $t_i[B_n] = t_j[B_n]$

FDs for Relational Schema Design

- High-level idea: **why do we care about FDs?**

1. Start with some relational *schema*

2. Find out its *functional dependencies (FDs)*

This part can be tricky!

3. Use these to *design a better schema*

1. One which minimizes possibility of anomalies

Finding Functional Dependencies

Equivalent to asking: Given a set of FDs, $F = \{f_1, \dots, f_n\}$, does an FD g hold?

Inference problem: How do we decide?

Answer: Three simple rules called **Armstrong's Rules**.

1. Split/Combine,
2. Reduction, and
3. Transitivity... *ideas by picture*

Closure of a set of Attributes

Given a set of attributes $A_1, \dots, A_n$ and a set of FDs F :

Then the closure, $\{A_1, \dots, A_n\}^+$ is the set of attributes B s.t. $\{A_1, \dots, A_n\} \rightarrow B$

Example: $F =$

$\{\text{name}\} \rightarrow \{\text{color}\}$
 $\{\text{category}\} \rightarrow \{\text{department}\}$
 $\{\text{color}, \text{category}\} \rightarrow \{\text{price}\}$

Example Closures:

$\{\text{name}\}^+ = \{\text{name}, \text{color}\}$
 $\{\text{name}, \text{category}\}^+ =$
 $\{\text{name}, \text{category}, \text{color}, \text{dept}, \text{price}\}$
 $\{\text{color}\}^+ = \{\text{color}\}$

Closure Algorithm

Start with $X = \{A_1, \dots, A_n\}$, FDs F .
Repeat until X doesn't change; **do**:
 if $\{B_1, \dots, B_n\} \rightarrow C$ is in F **and** $\{B_1, \dots, B_n\} \subseteq X$:
 then add C to X .
Return X as X^+

$F =$

$\{\text{name}\} \rightarrow \{\text{color}\}$

$\{\text{category}\} \rightarrow \{\text{dept}\}$

$\{\text{color, category}\} \rightarrow \{\text{price}\}$

$\{\text{name, category}\}^+ =$
 $\{\text{name, category}\}$

$\{\text{name, category}\}^+ =$
 $\{\text{name, category, color}\}$

$\{\text{name, category}\}^+ =$
 $\{\text{name, category, color, dept}\}$

$\{\text{name, category}\}^+ =$
 $\{\text{name, category, color, dept, price}\}$

Keys and Superkeys

A superkey is a set of attributes $A_1, \dots, A_n$ s.t. for *any other* attribute B in R , we have $\{A_1, \dots, A_n\} \rightarrow B$

I.e. all attributes are *functionally determined* by a superkey

A key is a *minimal* superkey

Meaning that no subset of a key is also a superkey

CALCULATING Keys and Superkeys

- **Superkey?**

- Compute the closure of A
- See if it = the full set of attributes

- **Key?**

- Confirm that A is superkey
- Make sure that no subset of A is a superkey
 - *Only need to check one 'level' down!*

Let A be a set of attributes, R set of all attributes, F set of FDs:

IsSuperkey(A, R, F):

$A^+ = \text{ComputeClosure}(A, F)$

Return $(A^+ == R)$?

IsKey(A, R, F):

If not *IsSuperkey*(A, R, F):

 return False

For B in *SubsetsOf*(A, size=len(A)-1):

 if *IsSuperkey*(B, R, F):

 return False

return True

Also see Lecture-5.ipynb!!!

High-Level: Lecture 7

- Conceptual design
- Boyce-Codd Normal Form (BCNF)
 - Definition
 - Algorithm
- Decompositions
 - Lossless vs. Lossy
 - A problem with BCNF
- MVDs
 - *In slightly greater depth since we skipped in lecture...*

Back to Conceptual Design

Now that we know how to find FDs, it's a straight-forward process:

1. Search for “bad” FDs
2. If there are any, then *keep decomposing the table into sub-tables* until no more bad FDs
3. When done, the database schema is *normalized*

Recall: there are several normal forms...

Boyce-Codd Normal Form

BCNF is a simple condition for removing anomalies from relations:

A relation R is in BCNF if:

if $\{A_1, \dots, A_n\} \rightarrow B$ is a *non-trivial* FD in R

then $\{A_1, \dots, A_n\}$ is a superkey for R

Equivalently: $\forall$ sets of attributes X , either $(X^+ = X)$ or $(X^+ = \text{all attributes})$

In other words: there are no “bad” FDs

Example

Name	SSN	PhoneNumber	City
Fred	123-45-6789	206-555-1234	Seattle
Fred	123-45-6789	206-555-6543	Seattle
Joe	987-65-4321	908-555-2121	Westfield
Joe	987-65-4321	908-555-1234	Westfield

$$\{SSN\} \rightarrow \{Name, City\}$$

This FD is *bad*
because it is not a
superkey

⇒ Not in BCNF

What is the key?
 $\{SSN, PhoneNumber\}$

Example

Name	<u>SSN</u>	City
Fred	123-45-6789	Seattle
Joe	987-65-4321	Madison

<u>SSN</u>	<u>PhoneNumber</u>
123-45-6789	206-555-1234
123-45-6789	206-555-6543
987-65-4321	908-555-2121
987-65-4321	908-555-1234

$\{SSN\} \rightarrow \{Name, City\}$

This FD is now
good because it is
the key

Let's check anomalies:

- Redundancy ?
- Update ?
- Delete ?

Now in BCNF!

BCNF Decomposition Algorithm

BCNFDecomp(R):

BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$
[all attributes]

Find a set of attributes X
which has non-trivial
“bad” FDs, i.e. is not a
superkey, using closures

BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$
[all attributes]

if (not found) then Return R

If no “bad” FDs found, in
BCNF!

BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$
[all attributes]

if (not found) then Return R

let $Y = X^+ - X$, $Z = (X^+)^C$

Let Y be the attributes that
 X functionally determines
(+ that are not in X)

And let Z be the other
attributes that it *doesn't*

BCNF Decomposition Algorithm

BCNFDecomp(R):

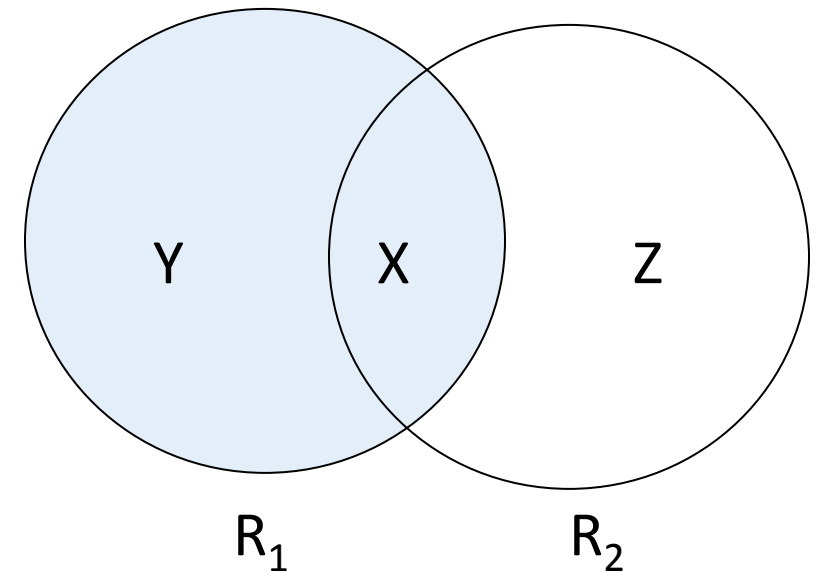
Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) then Return R

let $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

Split into one relation (table) with X plus the attributes that X determines (Y)...



BCNF Decomposition Algorithm

BCNFDecomp(R):

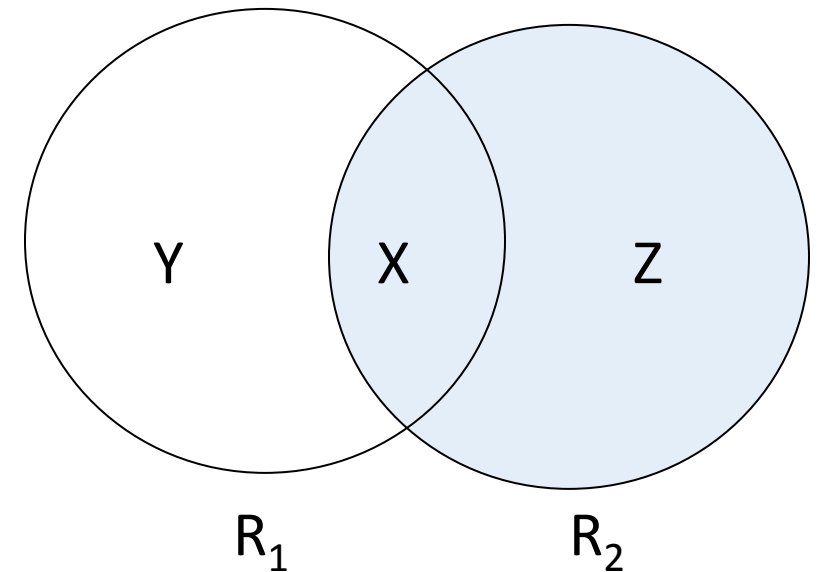
Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) then Return R

let $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

And one relation with X plus the attributes it *does not* determine (Z)



BCNF Decomposition Algorithm

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$
[all attributes]

if (not found) then Return R

let $Y = X^+ - X$, $Z = (X^+)^C$

decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

Proceed recursively until no
more “bad” FDs!

Example

BCNFDecomp(R):

Find a *set of attributes* X s.t.: $X^+ \neq X$ and $X^+ \neq$ [all attributes]

if (not found) then Return R

let $Y = X^+ - X$, $Z = (X^+)^c$

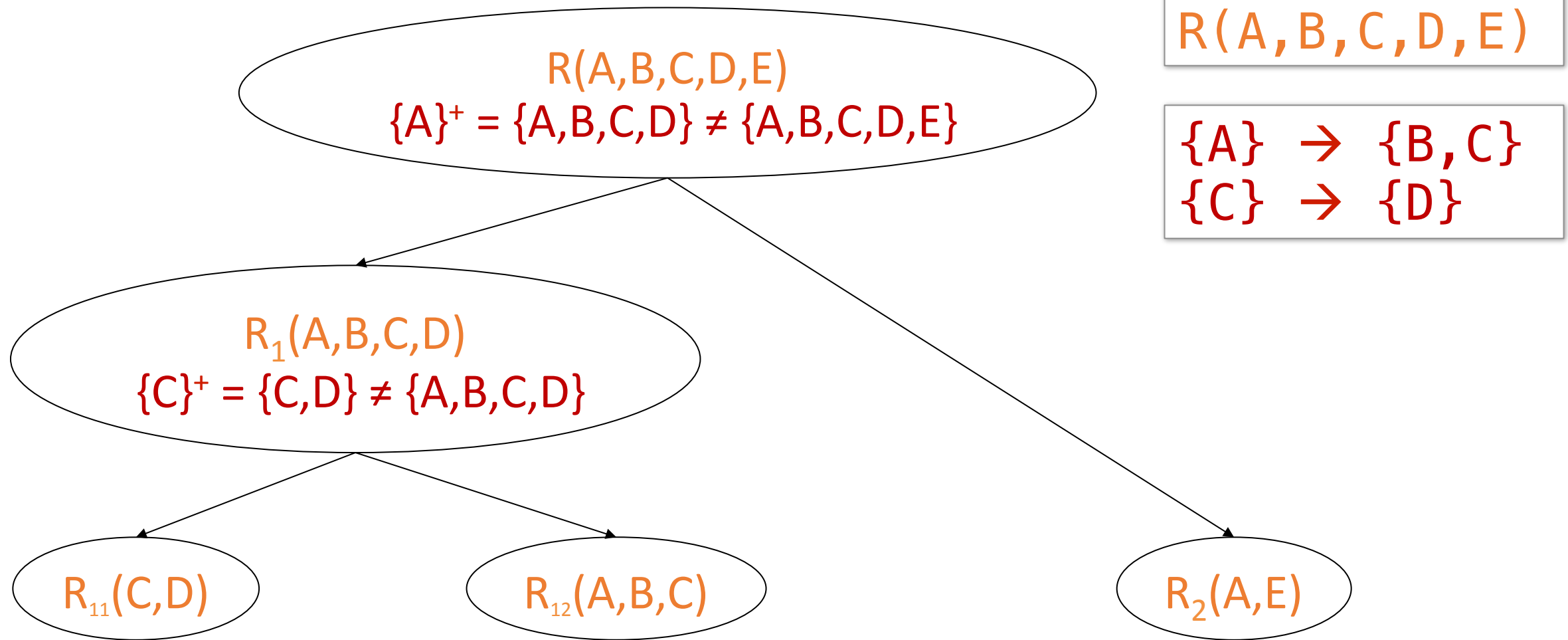
decompose R into $R_1(X \cup Y)$ and $R_2(X \cup Z)$

Return BCNFDecomp(R_1), BCNFDecomp(R_2)

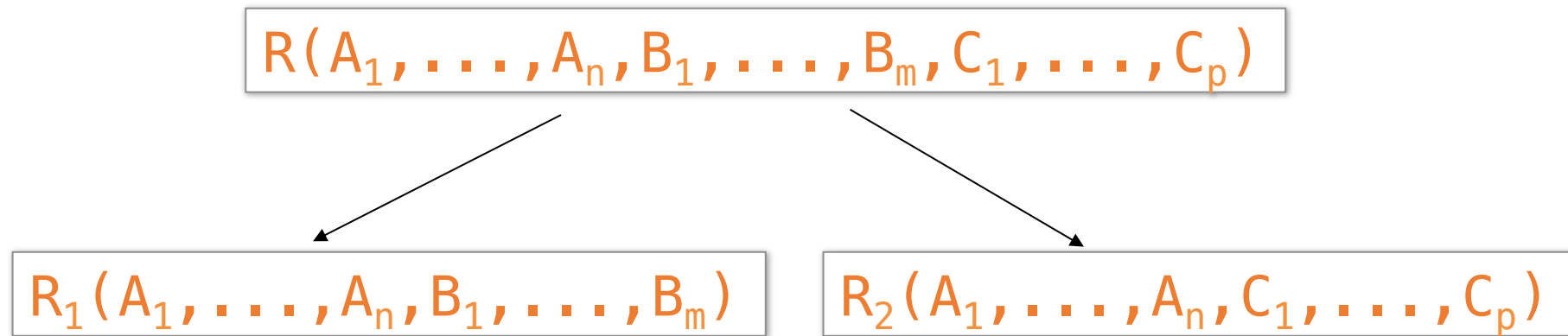
$R(A, B, C, D, E)$

$\{A\} \rightarrow \{B, C\}$
 $\{C\} \rightarrow \{D\}$

Example



Lossless Decompositions



If $\{A_1, \dots, A_n\} \rightarrow \{B_1, \dots, B_m\}$
Then the decomposition is lossless

Note: don't need
 $\{A_1, \dots, A_n\} \rightarrow \{C_1, \dots, C_p\}$

BCNF decomposition is always lossless. Why?

A Problem with BCNF

Unit	Company	Product
...	...	...

<u>Unit</u>	Company
...	...

Unit	Product
...	...

$$\{Unit\} \rightarrow \{Company\}$$

$$\begin{aligned} \{Unit\} &\rightarrow \{Company\} \\ \{Company, Product\} &\rightarrow \{Unit\} \end{aligned}$$

We do a BCNF decomposition
on a “bad” FD:
 $\{Unit\}^+ = \{Unit, Company\}$

We lose the FD $\{Company, Product\} \rightarrow \{Unit\}!!$

Multiple Value Dependencies (MVDs)

Many of you asked, “what do these mean in real life?”



Grad student CA thinks:
“Hmm... what is real life??
Watching a movie over the
weekend?”

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

Are there any functional dependencies that might hold here?

No...

And yet it seems like there is some pattern / dependency...

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre...

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre...

Given a set of movies and snacks...

MVDs: Movie Theatre Example

Movie_theater	film_name	snack
Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
Rains 216	Star Trek: The Wrath of Kahn	Burrito
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
Rains 218	Star Wars: The Boba Fett Prequel	Ramen
Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

For a given movie theatre...

Given a set of movies and snacks...

Any movie / snack combination is possible!

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $T_3[A] = t_1[A]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_1	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
t_3	Rains 216	Star Trek: The Wrath of Kahn	Burrito
	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_2	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

More formally, we write $\{A\} \twoheadrightarrow \{B\}$ if for any tuples t_1, t_2 s.t. $t_1[A] = t_2[A]$ there is a tuple t_3 s.t.

- $t_3[A] = t_1[A]$
- $t_3[B] = t_1[B]$
- and $t_3[R \setminus B] = t_2[R \setminus B]$

Where $R \setminus B$ is “R minus B” i.e. the attributes of R not in B

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t_2	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
t_3	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t_1	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

Note this also works!

Remember, an MVD holds over *a relation or an instance*, so defn. must hold for every applicable pair...

MVDs: Movie Theatre Example

	Movie_theater (A)	film_name (B)	Snack (C)
t ₂	Rains 216	Star Trek: The Wrath of Kahn	Kale Chips
	Rains 216	Star Trek: The Wrath of Kahn	Burrito
t ₃	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Kale Chips
t ₁	Rains 216	Lord of the Rings: Concatenated & Extended Edition	Burrito
	Rains 218	Star Wars: The Boba Fett Prequel	Ramen
	Rains 218	Star Wars: The Boba Fett Prequel	Plain Pasta

This expresses a sort of dependency (= data redundancy) that we *can't* express with FDs

** Actually, it expresses conditional independence (between film and snack given movie theatre)!*

MVDs...

Think you can't understand them?

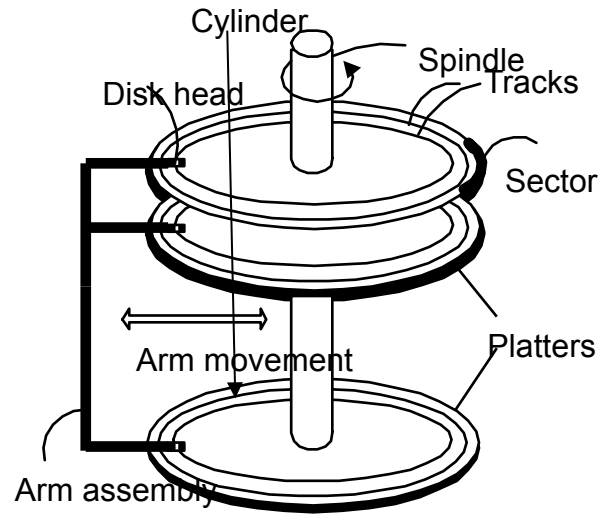
YES YOU



High-Level: Lecture 8

- Our model of the computer: Disk vs. RAM, local vs. global
- Transactions (TXNs)
- ACID
- Logging for Atomicity & Durability
 - Write-ahead logging (WAL)

High-level: Disk vs. Main Memory



Disk:

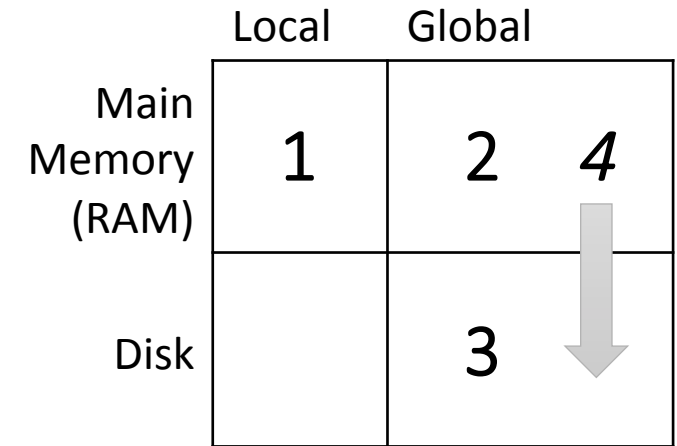
- **Slow:** Sequential access
 - (although fast sequential reads)
- **Durable:** We will assume that once on disk, data is safe!
- **Cheap**

Random Access Memory (RAM) or Main Memory:

- **Fast:** Random access, byte addressable
 - ~10x faster for sequential access
 - ~100,000x faster for random access!
- **Volatile:** Data can be lost if e.g. crash occurs, power goes out, etc!
- **Expensive:** For \$100, get 16GB of RAM vs. 2TB of disk!

Our model: Three Types of Regions of Memory

- 1. Local:** In our model each process in a DBMS has its own local memory, where it stores values that only it “sees”
- 2. Global:** Each process can read from / write to shared data in main memory
- 3. Disk:** Global memory can read from / flush to disk
- 4. Log:** Assume on stable disk storage- spans both main memory and disk...



Log is a *sequence* from main memory -> disk

“Flushing to disk” = writing to disk + erasing (“evicting”) from main memory

Transactions: Basic Definition

A transaction (“TXN”) is a sequence of one or more *operations* (reads or writes) which reflects *a single real-world transition*.

In the real world, a TXN either happened completely or not at all

```
START TRANSACTION
UPDATE Product
SET Price = Price - 1.99
WHERE pname = 'Gizmo'
COMMIT
```

Transaction Properties: ACID

- **A**tomic
 - State shows either all the effects of txn, or none of them
- **C**onsistent
 - Txn moves from a state where integrity holds, to another where integrity holds
- **I**solated
 - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **D**urable
 - Once a txn has committed, its effects remain in the database

ACID is/was source of great debate!

Goal of LOGGING: Ensuring Atomicity & Durability

ACID

- **Atomicity:**
 - TXNs should either happen completely or not at all
 - If abort / crash during TXN, *no* effects should be seen
- **Durability:**
 - If DBMS stops running, changes due to completed TXNs should all persist
 - *Just store on stable disk*

TXN 1



Crash / abort

No changes
persisted

TXN 2



All changes
persisted

Basic Idea: (Physical) Logging

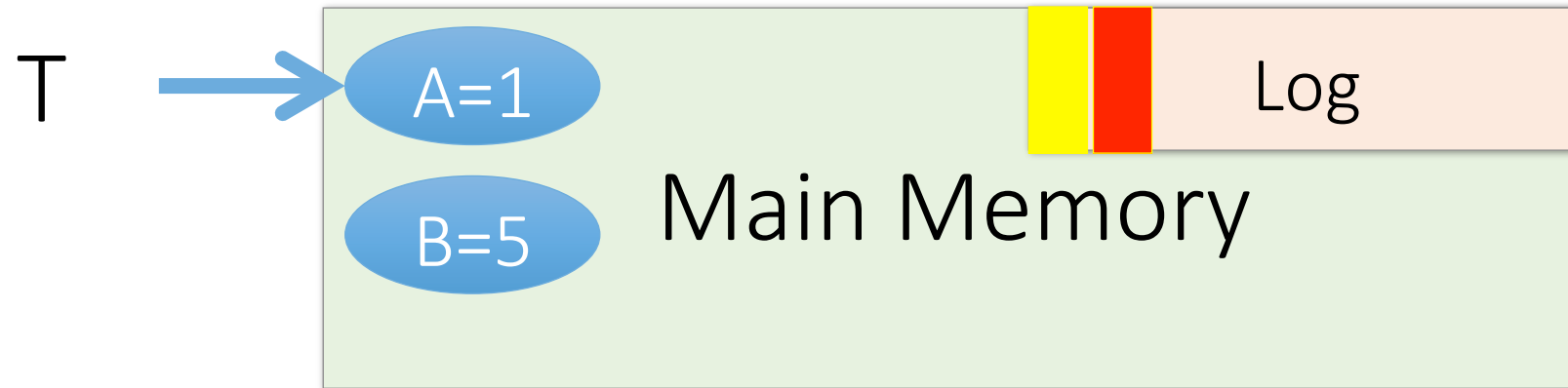
- Record UNDO information for every update!
 - Sequential writes to log
 - Minimal info (diff) written to log
- The **log** consists of **an ordered list of actions**
 - Log record contains:
<XID, location, old data, new data>

This is sufficient to UNDO any transaction!

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

A: 0 → 1



This time, let's try committing after we've written log to disk but before we've written data to disk... this is WAL!

OK, Commit!

If we crash now, is T durable?



Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)



A: 0 → 1



This time, let's try committing after we've written log to disk but before we've written data to disk... this is WAL!

OK, Commit!

If we crash now, is T durable?

USE THE LOG!

Write-Ahead Logging (WAL)

- DB uses **Write-Ahead Logging (WAL)** Protocol:

Each update is logged! Why not reads?

1. Must *force log record* for an update *before* the corresponding data page goes to storage

→ Atomicity

2. Must *write all log records* for a TX *before commit*

→ Durability

High-Level: Lecture 9

- Motivation: Concurrency with Isolation & consistency
 - Using TXNs...
- Scheduling
- Serializability
- Conflict types & classic anomalies

Concurrency: Isolation & Consistency

- The DBMS must handle concurrency such that...

1. **Isolation** is maintained: Users must be able to execute each TXN **as if they were the only user**

- DBMS handles the details of *interleaving* various TXNs

ACID

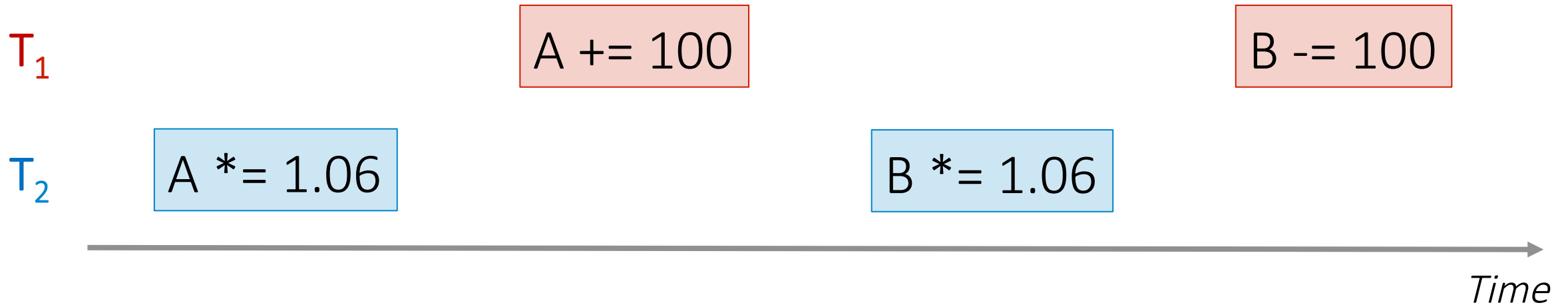
2. **Consistency** is maintained: TXNs must leave the DB in a **consistent state**

- DBMS handles the details of enforcing integrity constraints

ACCD

Example- consider two TXNs:

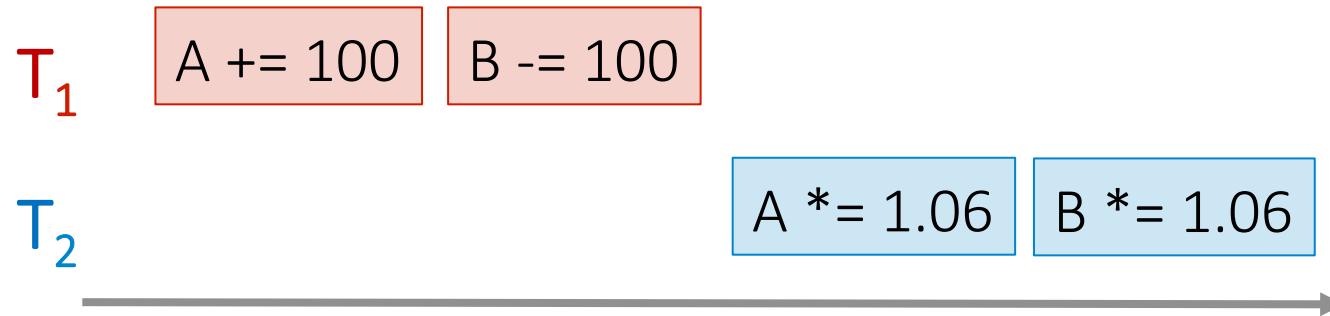
The DBMS can also **interleave** the TXNs



What goes / could go wrong here??

Scheduling examples

Serial schedule $T_1 \rightarrow T_2$:

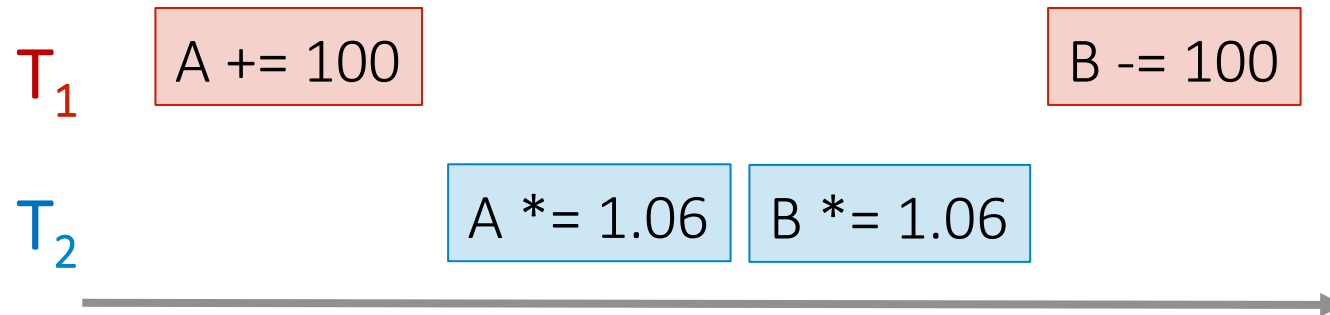


*Starting
Balance*

A	B
\$50	\$200

A	B
\$159	\$106

Interleaved schedule B:



A	B
\$159	\$112

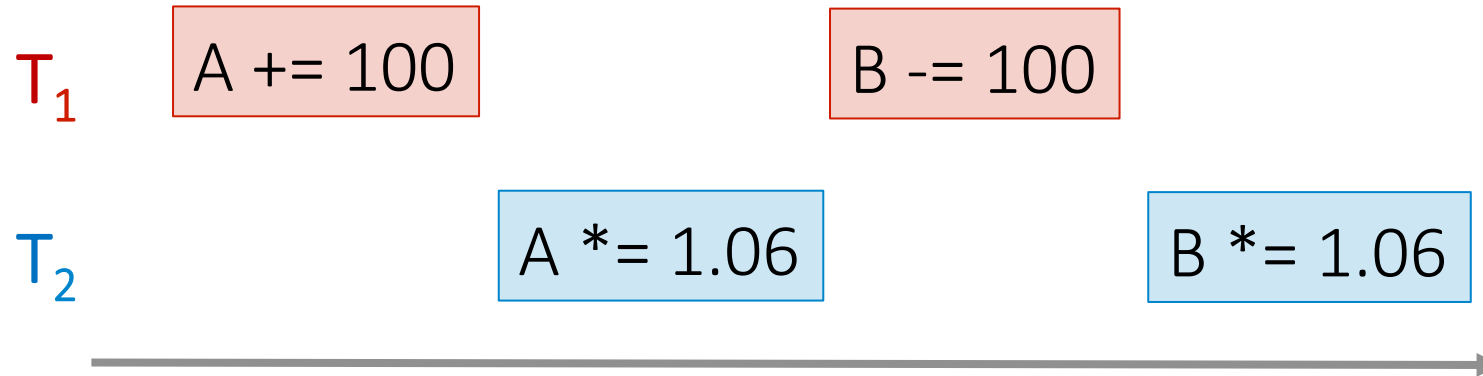
Different
result than
serial
 $T_1 \rightarrow T_2!$

Scheduling Definitions

- A **serial schedule** is one that does not interleave the actions of different transactions
- A and B are **equivalent schedules** if, *for any database state*, the effect on DB of executing A is **identical to** the effect of executing B
- A **serializable schedule** is a schedule that is equivalent to ***some*** serial execution of the transactions.

The word “**some**” makes this def powerful and tricky!

Serializable?



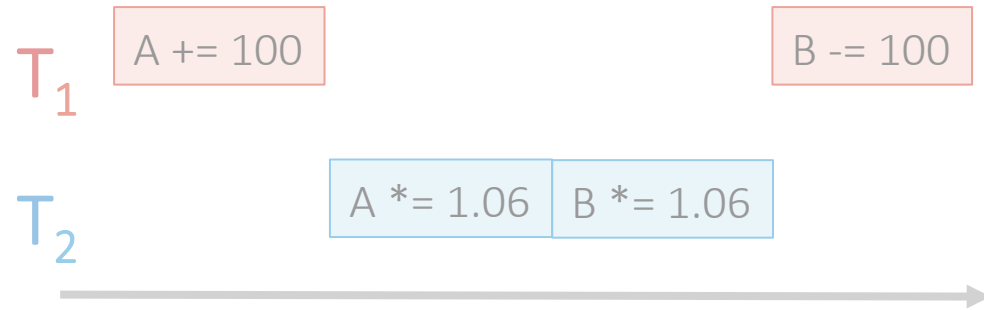
Serial schedules:

	A	B
$T_1 \rightarrow T_2$	$1.06 * (A + 100)$	$1.06 * (B - 100)$
$T_2 \rightarrow T_1$	$1.06 * A + 100$	$1.06 * B - 100$

A	B
$1.06 * (A + 100)$	$1.06 * (B - 100)$

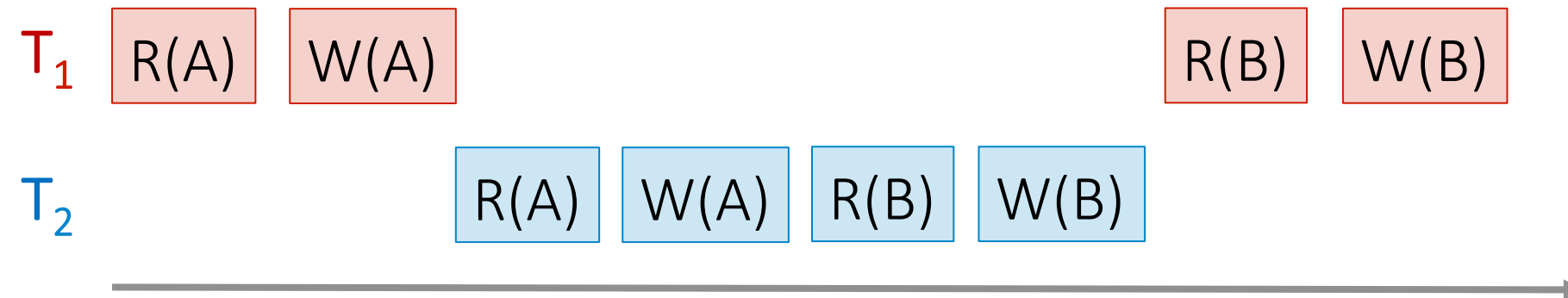
Same as a serial schedule
for all possible values of
 $A, B = \underline{\text{serializable}}$

The DBMS's view of the schedule



Each action in the TXNs
*reads a value from global
memory and then writes
one back to it*

Scheduling order matters!



Conflict Types

Two actions conflict if they are part of different TXNs, involve the same variable, and at least one of them is a write

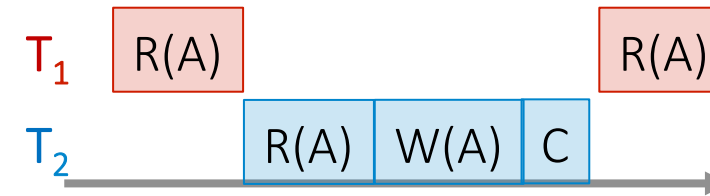
- Thus, there are three types of conflicts:
 - Read-Write conflicts (RW)
 - Write-Read conflicts (WR)
 - Write-Write conflicts (WW)

Why no “RR Conflict”?

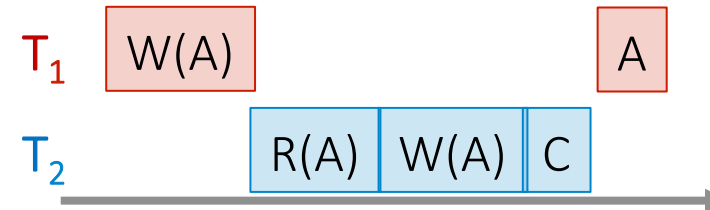
Interleaving anomalies occur with / because of these conflicts between TXNs *(but these conflicts can occur without causing anomalies!)*

Classic Anomalies with Interleaved Execution

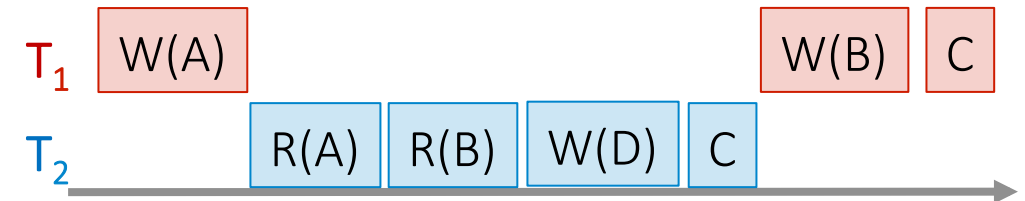
“Unrepeatable read”:



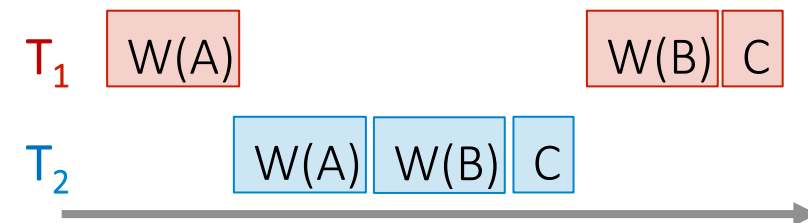
“Dirty read” / Reading uncommitted data:



“Inconsistent read” / Reading partial commits:



Partially-lost update:

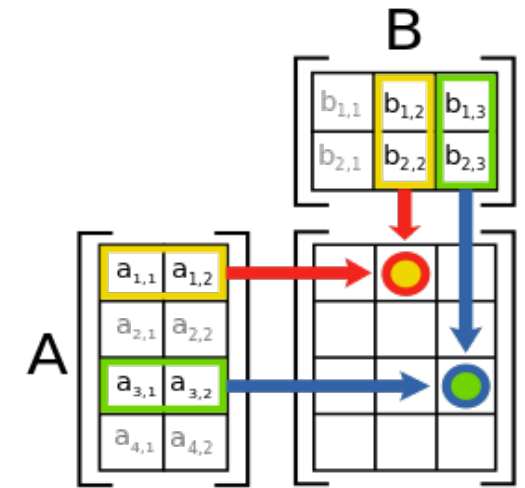


Notes

- Locking & etc. (all content in lecture 9 after activity 9-1) will **not** be covered
- PS1 review slides included as appendix (after this...)
- PS2 & additional content covered in extra review session on **Sunday, in Gates 104**

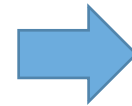
Linear Algebra, Declaratively

- Matrix multiplication & other operations = just **joins**!
- The shift from **procedural** to **declarative** programming



$$C[i][j] = \sum_{k=1}^m A[i][k] * B[k][j]$$

```
C = [[0]*p for i in range(n)]
for i in range(n):
    for j in range(p):
        for k in range(m):
            C[i][j] += A[i][k] * B[k][j]
```



```
SELECT A.i, B.j, SUM(A.x * B.x)
FROM A, B
WHERE A.j = B.i
GROUP BY A.i, B.j;
```

Proceed through a series of instructions

Declare a desired output set

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,  
       COUNT(day) AS nbd  
FROM precipitation,  
     (SELECT day, MAX(precip)  
      FROM precipitation  
      GROUP BY day) AS m  
WHERE day = m.day AND precip = m.precip  
GROUP BY station_id  
HAVING COUNT(day) > 1  
ORDER BY nbd DESC;
```

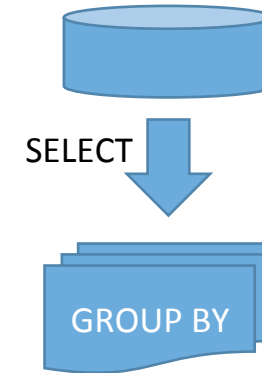
Think about **order***!

**of the semantics, not the
actual execution*

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,  
       COUNT(day) AS nbd  
FROM precipitation,  
     (SELECT day, MAX(precip)  
      FROM precipitation  
      GROUP BY day) AS m  
WHERE day = m.day AND precip = m.precip  
GROUP BY station_id  
HAVING COUNT(day) > 1  
ORDER BY nbd DESC;
```

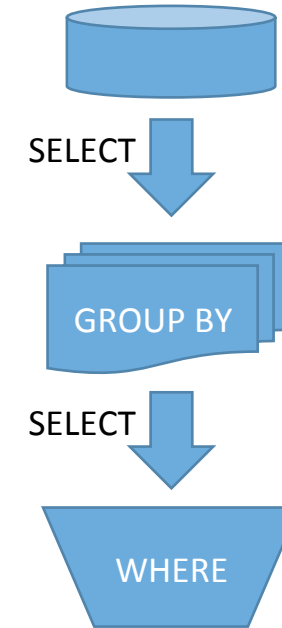


Get the max
precipitation **by day**

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,  
       COUNT(day) AS nbd  
FROM precipitation,  
     (SELECT day, MAX(precip)  
      FROM precipitation  
      GROUP BY day) AS m  
WHERE day = m.day AND precip = m.precip  
GROUP BY station_id  
HAVING COUNT(day) > 1  
ORDER BY nbd DESC;
```



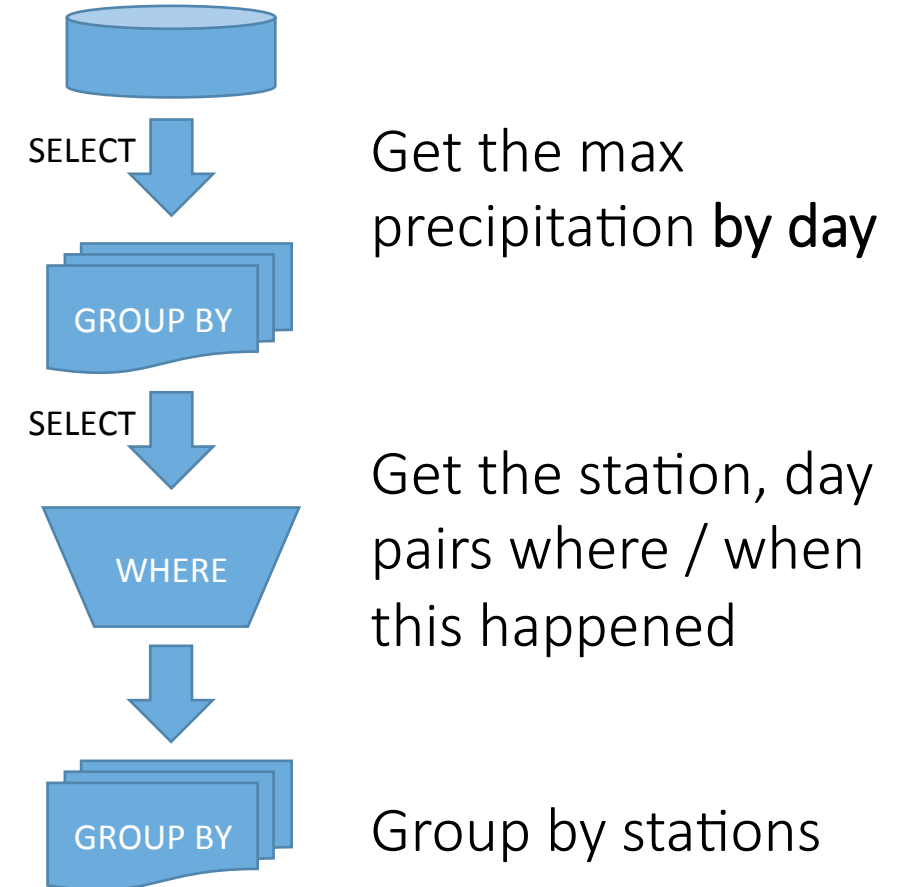
Get the max
precipitation **by day**

Get the station, day
pairs where / when
this happened

Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

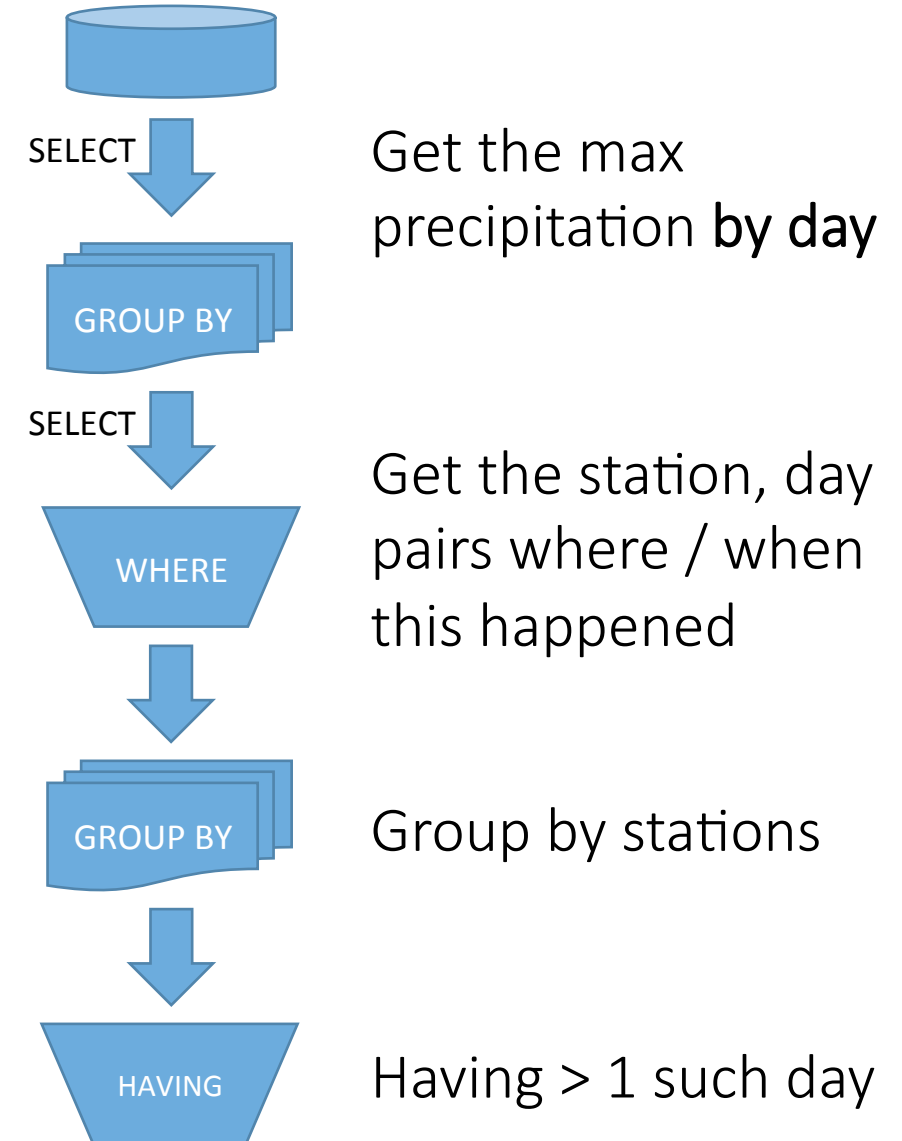
```
SELECT station_id,  
       COUNT(day) AS nbd  
FROM precipitation,  
     (SELECT day, MAX(precip)  
      FROM precipitation  
      GROUP BY day) AS m  
WHERE day = m.day AND precip = m.precip  
GROUP BY station_id  
HAVING COUNT(day) > 1  
ORDER BY nbd DESC;
```



Common SQL Query Paradigms

GROUP BY / HAVING + Aggregators + Nested queries

```
SELECT station_id,  
       COUNT(day) AS nbd  
FROM precipitation,  
     (SELECT day, MAX(precip)  
      FROM precipitation  
      GROUP BY day) AS m  
WHERE day = m.day AND precip = m.precip  
GROUP BY station_id  
HAVING COUNT(day) > 1  
ORDER BY nbd DESC;
```



Common SQL Query Paradigms

Complex correlated queries

```
SELECT x1.p AS median
FROM x AS x1
WHERE
    (SELECT COUNT(*)
     FROM X AS x2
     WHERE x2.p > x1.p)
    =
    (SELECT COUNT(*)
     FROM X AS x2
     WHERE x2.p < x1.p);
```

This was a tricky problem- but good practice in thinking about things declaratively

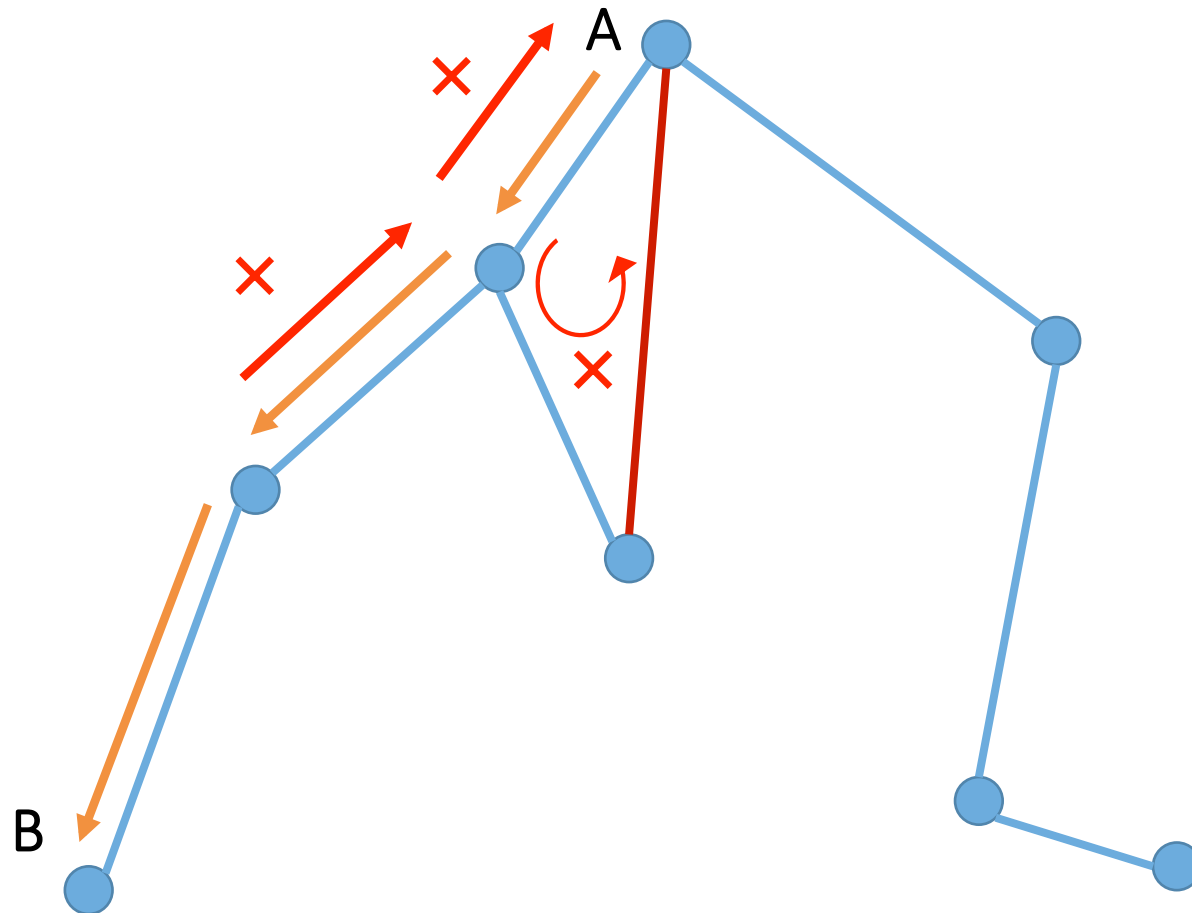
Common SQL Query Paradigms

Nesting + EXISTS / ANY / ALL

```
SELECT sid, p3.precip
FROM (
    SELECT sid, precip
    FROM precipitation AS p1
    WHERE precip > 0 AND NOT EXISTS (
        SELECT p2.precip
        FROM precipitation AS p2
        WHERE p2.sid = p1.sid
            AND p2.precip > 0
            AND p2.precip < p1.precip)) AS p3
WHERE NOT EXISTS (
    SELECT p4.precip
    FROM precipitation AS p4
    WHERE p4.precip - 400 > p3.precip);
```

More complex,
but again just
think about
order!

Graph traversal & recursion

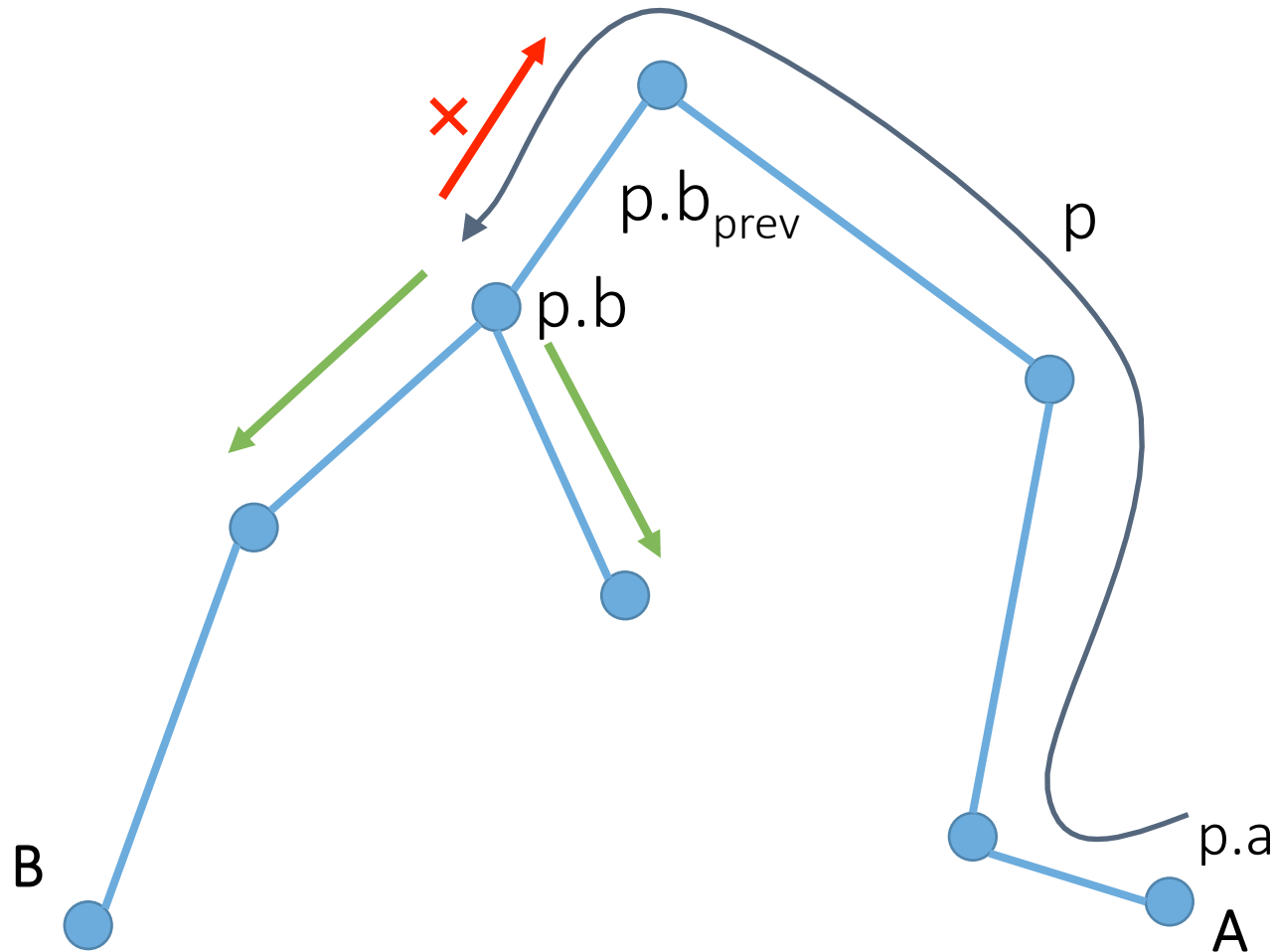


For fixed-length paths

```

SELECT A, B, d
FROM edges
UNION
SELECT e1.A, e2.B,
       e1.d + e2.d AS d
FROM edges e1, edges e2
WHERE e1.B = e2.A
      AND e2.B <> e1.A
UNION
SELECT e1.A, e3.B,
       e1.d + e2.d + e3.d AS d
FROM edges e1, edges e2, edges e3
WHERE e1.B = e2.A
      AND e2.B = e3.A
      AND e2.B <> e1.A
      AND e3.B <> e2.A
      AND e3.B <> e1.A
  
```

Graph traversal & recursion



For variable-length paths on trees

```
WITH RECURSIVE
paths(a, b, b_prev, d) AS (
  SELECT A, B, A
  FROM edges
  UNION
  SELECT p.a, e.B, e.A,
         p.d + e.d
  FROM paths p, edges e
  WHERE p.b = e.A
        AND e.B <> p.b_prev)
SELECT a, b, MAX(d)
FROM paths;
```